Chapter 50

Implementing Dynamic Programming

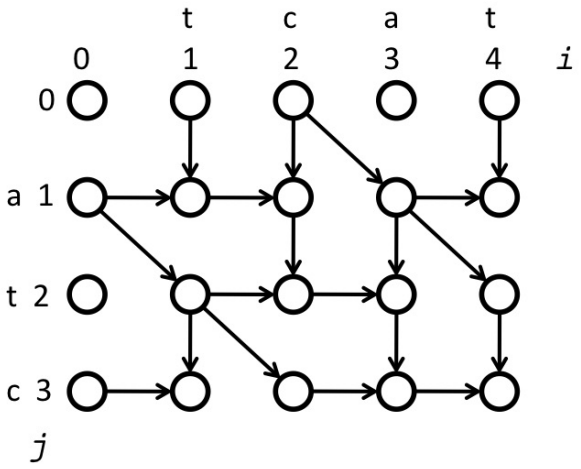
In previous chapters, we have considered several problems and showed that these problems admit a recursive solution, which via sharing, can be computed in polynomial work. Without sharing, the recursive solutions would require exponential work. In this chapter, we present two techniques for implementing such sharing. The two techniques called, bottom-up , and top-down , are similar but different enough to have each their own advantages.

1 Bottom-Up Method

Constructing the DAG Bottom-Up. Given the DAG of a recursive solution, observe that the leaves of the DAG may be computed directly because they don't depend on any other results. As we compute each vertex, imagine "pebbling" the vertex so as to indicate that it is now available. Now, once the leaves are pebbled, any other vertex of the DAG, all of whose in-neighbors are pebbled may also be computed and pebbled. This process of pebbling a vertex whose in-neighbors are pebbled may be continued until all vertices of the DAG, including the root, are pebbled. The pebbling of the root, in turn, completes the computation and yields the result.

Depending on the topology of the DAG, there is usually plenty of freedom about which vertex or vertices, we can pebble. Often times, there are not just one but many vertices, all of which may be pebbled at once, i.e., in parallel. Depending on our goals, we map prefer one pebbling strategy over another, to optimize for various metrics such as work, space, and parallelism.

Example 50.1 (Minimum Edit Distance)**.** Consider Minimum Edit Distance problem for the two strings $S = \texttt{tcat}$ and $T = \texttt{atc}$. We can draw the DAG as follows.



In the DAG, there are three types of edges

1. down edges
2. horizontal edges, which go from left to right, and
3. diagonal edges.

The numbers represent the i and the j for that position in the string. We draw the DAG with the root at the bottom right, so that the vertices are structured the same way we might fill an array indexed by i and j.

As an example, consider $MED(4, 3)$. The characters $S[4]$ and $T[3]$ are not equal so the recursive calls are to $MED(3, 3)$ and $MED(4, 2)$. This corresponds to the vertex to the left and the one above. Now if we consider $MED(4, 2)$ the characters $S[4]$ and $T[2]$ are equal so the recursive call is to $MED(3, 1)$. This corresponds to the vertex diagonally above and to the left. In fact whenever the characters $S[i]$ and $T[j]$ are not equal we have edges from directly above and directly to the left, and whenever they are equal we have an edge from the diagonal to the left and above.

Based on the direction of the edges, we can pebble the vertices of the DAG in several different ways. For example, we can pebble the DAG row-wise, by pebbling the first row from left to right, and then the second row, and so on. Symmetrically, we can pebble the DAG column-wise, starting with the first column and pebbling from top to bottom, and then proceeding to the second and so on. The row-wise and column-wise pebbling orders both yield a sequential algorithm. Finally, we can pebble the dag diagonally from top left towards bottom right. In this order, each position within a diagonal may be pebbled in parallel, leading thus to a parallel algorithm.

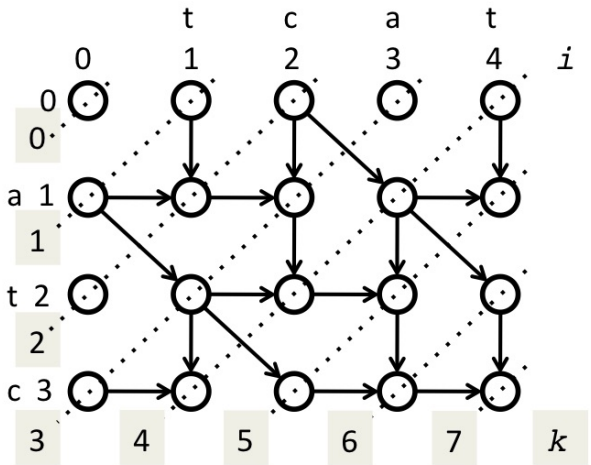
Algorithm 50.1 (Bottom up MED)**.** The pseudo-ccode for the bottom-up algorithm for MED is given below. This algorithm pebbles the DAG diagonally and stores the result of each vertex in a table M. Because the table is indexed by two integers, it can be represented by an array, which allows constant work random access. Each call to $diagonals$

processes one diagonal and updates the table M. The size of the diagonals grows and then shrinks. We note that the index calculations are tricky.

```
med S T =
  let
    medOne M (i, j) =
      case (i, j) of
        (i, 0) ⇒ i
      | (0, j) ⇒ j
      | (i, j) ⇒
          if (S[i − 1] = T[j − 1]) then M[i − 1, j − 1]
          else 1 + min(M[i, j − 1], M[i − 1, j])

    diagonals M k =
      if (k > |S| + |T|) then
        M
      else
        let
          s = max(0, k − |T|)
          e = min(k, |S|)
          MM = M ∪ {(i, k − i) ↦ medOne M (i, k − i) : i ∈ {s, . . . , e}}
        in
          diagonals MM(k + 1)
        end
  in
    diagonals {} 0
  end
```

Example 50.2. The drawing below illustrates the diagonals as pebbled by the bottom-up algorithm given above.



2 Top-Down Method: Memoization

Definition 50.2 (Memoization and Memo Table)**.** The top-down approach is based on running the recursive code pretty much as is, but with some additional structure to store results as they are computed. In particular, we keep a mapping from the arguments of the recursive function to its solutions, and each time we return from a recursive call, we add the argument-result pair to the mapping. This way when we come across the same argument a second time we can just look up the solution instead of having to recompute it. This process is called ***memoization***, and the table used to map the arguments to solutions is called a ***memo table***.

Implementing Memoization. Being able to store and look up previous results efficiently is key to effective memoization. This in turn requires being able to check quickly if an argument is equal to a previous argument. If the arguments are complex data structures then such an equality test might be expensive on its own. Beyond checking for equality, looking up a previous result also requires storing the mapping in a table in which insertions and lookups are fast. Obvious choices are either balanced search trees or hash tables. The first requires a total order over the possible arguments, and an efficient way to check if one argument is "less" than another. The second requires that the arguments can be hashed, and to be efficient we need that it is unlikely that two different arguments hash to the value.

Checking equality of arbitrary arguments, comparing them, or hashing them can be complicated. Fortunately in many cases, arguments can be arranged so that they are integers or small sequences of integers (tuples, triples, etc). Such integer-valued arguments simplify the implementation of such operations: we can cheaply compare or hash integer arguments.

Example 50.3. For the examples that we have consider so far including the Subset Sum Problem , the Minimum Edit Distance Problem , and the Optimal Binary Search Problem , we have set up *surrogate* integer values to represent the input values.

Algorithm 50.3 (The Memo Function)**.** To implement the memoization we define the following function:

```
memo f M a =
  case find M a of
    SOME(v) ⇒ (M, v)
  | NONE ⇒  let (M', v) = f M a
            in (update(M', a, v), v) end
```

In this function f is the function that is being memoized, M is the memo table, and a is the argument to f. This function simply looks up the value a in the memo table. If it exists, then it returns the corresponding result. Otherwise it evaluates the function on the argument, and as well as returning the result it stores it in the memo.

As an example, we can now write med using memoization as follows.

Algorithm 50.4 (Memoized MED)**.** The pseudo-code below uses memoization to achieve sharing of solutions to subproblems.

```
med S T =
  let
    medOne M (i, j) =
      case (i, j) of
        (_, 0) ⇒ (M, i)
      | (0, _) ⇒ (M, j)
      | _ ⇒  if (S[i − 1] = T[i − 1]) then
               memo medOne M (i − 1, j − 1)
             else
               let
                 (M2, v1) = memo medOne M (i, j − 1)
                 (M3, v2) = memo medOne M2 (i − 1, j)
               in
                 (M3, 1 + min(v1, v2))
               end
    (_, r) = medOne {} (|S|, |T|))
  in
    r
  end
```

Note. Note that the memo table M is threaded through the algorithm. In particular every call to MED takes a memo table as an argument, and returns a memo table as a result (possibly updated). Because of this passing, the code is purely functional.

Limitation of Top-Down Method. The top-down approach as described is is inherently sequential. By threading the memo table through the computation, we force a total ordering on all calls to med. It is possible to solve these problems by combining several techniques

- using hidden state to implement the $memo$ function such that the memo table is used implicitly
- using concurrent hash tables to store the results so that parallel calls might be in flight at the same time, and
- using synchronization variables to make sure that no function is computed more than once.

These techniques are all advanced techniques and are beyond the scope of this book.